Device for the photorealistic representation of dynamic, complex, three-dimensional scenes by means of ray tracing

The invention relates to a device with which dynamic, complex, three-dimensional scenes can be represented with high image repetition rates on a two-dimensional display through use of real-time ray-tracing hardware architecture. Dynamic scenes are scenes in which, aside from the camera position, the geometry of the objects to be represented can change from frame to frame. The invention is characterized above all by the fact that it supports a hierarchical object structure, that is, the main scene may consist of a plurality of objects each of which is built up from sub-objects; this nesting may be extended to an arbitrary depth. The objects on any one hierarchy level may be moved individually but also together. This makes it possible to create highly dynamic, complex scenes and, by using the same object at a multiple number of locations in the scene, to keep the representation of the scene in the memory small.

These nested object levels are realized according to the invention by extending the hardware implementation of the known ray-tracing pipeline by a transformation unit included in the hardware, said transformation unit transforming the rays into the objects. In order to make optimal use of the hardware resources, this unit is only included once; aside from object space transformation, it is used to compute the ray-triangle intersection points, to generate primary rays and to generate secondary rays.

Through use of special processors designed for ray tracing, the invention allows the user the full system programmability by using, according to the invention, a novel processor architecture consisting of the combination of a standard processor core with one or more special ray-tracing instructions. Use of these ray-tracing processors permits the programming of a wide variety of ray-tracing procedures. Primitive scene objects can be configured programmably, so that, in contrast to today's graphic cards, the use of spline surfaces is also possible; to this end, a special algorithm is programmed for computing the intersection of a ray with the spline surface. As is standard in today's rasterization hardware, a wide variety of surface shading models may be programmed.

To output the picture data on a display, the invention may be combined with state-of-the-art rasterization hardware by using shared frame buffers and z-buffers.

Prior art

The prior art relating to the representation of three-dimensional scenes currently falls under two main categories, namely rasterization and ray-tracing (see Computer Graphics / Addison-Wesley ISBN 0201848406).

The well-known rasterization process, which is used primarily in computer graphic cards, is based on the principle of projecting every scene geometry onto a frame buffer and z-buffer. To this end, the color and brightness values of the pixels are stored in the frame buffer and the geometric depth values in the z-buffer, but only if the previous geometric value in the z-buffer is greater (further away from the viewer) than the new one. In this way, it is ensured that closer objects overwrite more distant ones, and that at the end of the process, only the actually visible objects are imaged in the frame buffer.

However, this method has the serious disadvantage that complex scenes involving millions of objects cannot be represented in real time with the hitherto known hardware, because, as a rule, it is necessary to project all the triangles (objects) of the scene. Furthermore, the process requires a frame buffer and a z-buffer, on which many billions of write operations must be performed per second; for image formatting, most of the pixels are overwritten several times per frame. As a result of pixels further from the viewer being overwritten by pixels of closer objects, already computed data are discarded, thus preventing optimal system performance.

Shadows can be computed with today's rasterization hardware by employing complex techniques, but accuracy problems are encountered with complex scenes. Neither specular reflections at curved surfaces nor the computation of light refractions can be realized physically correctly with this technique.

System performance is enhanced by a second method, the ray-tracing process, which is well known for its photorealistic images but also for its computational complexity. The basic idea behind ray tracing is closely related to physical light distribution models (see Computer Graphics / Addison-Wesley ISBN 0201848406).

In a real environment, light is emitted from light sources and is distributed in the scene according to physical laws. The picture of the environment can be captured by a camera.

Ray tracing works the opposite way round, and traces the light from the camera, which represents the viewer's position, back to its source. This entails shooting a virtual ray for each pixel of the image in the direction illuminating the pixel. This shooting of the ray is called ray casting. If the ray strikes an object, the color of the pixel is computed from, among other things, the color of the object encountered, the normals to the surface and the light sources visible from the point of impact. The visible light sources are determined by tracing secondary rays shot from each light source to the point of impact. If these shadow rays encounter an object between the light source and the point of impact, the point is in the shadow with respect to the light source.

Aside from the described shadow computation, this method also allows computation of specular reflections and of light refractions by means of computing reflection rays and refracted secondary rays. An added advantage is that scenes of almost arbitrary size can be handled and represented. The reason for this is that an acceleration structure is used. This is a special process with an appropriate data structure that makes it possible to “shoot” and traverse the virtual ray rapidly through the scene. A number of objects that are potential hit candidates are selected on the way, as a result of which the point of impact is quickly found. Theoretical studies have shown that on average, the complexity of the ray tracing process grows logarithmically with the size of the scene. That means that squaring the number of scene objects only doubles the computational overhead.

Typical acceleration structures are, for example, the uniform grid, the kD tree, the octree and the bounding-volume hierarchy (see Computer Graphics / Addison-Wesley ISBN 0201848406). All these techniques are based on the idea of subdividing the space into many cells and storing the geometry of each of these cells. The traversal process then traces the ray from cell to cell and always intersects it with precisely those objects that are located in the cell. The four techniques differ only in the method of subdivision. With the uniform grid, the space is subdivided into cube-shaped cells of equal size. The drawing of Figure 7 illustrates this technique. The three other techniques are based on recursive space subdivision. In the kD-tree technique, the starting space is subdivided recursively and axis-parallel at an arbitrary point. The drawing of Fig. 8 illustrates this technique. This subdivision of the space is stored in a recursive data structure (a binary tree). The third technique, called octree, is likewise recursive, the only difference being that the relevant cells are always subdivided into 8 equal-sized rectangular cells. This is illustrated in Fig. 9. The bounding-volume hierarchy subdivides the space into n arbitrary volumes, which, unlike in the other techniques, are even allowed to overlap.

In contrast to the rasterization process, there is currently no pure hardware solution that implements the ray-tracing process, but only software-based systems that need a relatively large amount of computational power and computing time. To illustrate the extent of time required for the computations it may be remarked that with PC hardware conforming to the current prior art, a computation time of several seconds to several hours – the exact time will depend on the complexity of the image – is needed to create a single still image using this method. The computation of moving images requires a correspondingly large amount of time and/or the availability of special mainframes.

The Department of Computer Graphics at the University of Saarland has developed a software-based real-time ray-tracing system that is used on a cluster of over 20 computers.

The US patent 6,597,359 B1 describes a hardware solution for ray tracing, but it is limited to static scenes.

The US patent 5,933,146 likewise describes a hardware solution for ray tracing, which is likewise limited to static scenes.

The paper "SaarCOR - A Hardware Architecture for Ray-Tracing" by the Department of Computer Graphics at the University of Saarland describes a hardware architecture for ray tracing, but it is again limited to static scenes.

The paper "A Simple and Practical Method for Interactive Ray-Tracing of Dynamic Scenes" by the Department of Computer Graphics at the University of Saarland describes a software approach to supporting dynamic scenes in a ray tracer. However, the software process described uses only one object level, i.e. it cannot handle multi-level nesting.

The described prior art currently offers neither software nor hardware solutions with which complex dynamic scenes can be represented in real time. The performance limitation of the known rasterization processes lies in the number of objects to be represented.

Ray-tracing systems are admittedly able to represent numerous triangles, but, on account of the preliminary computations that are necessary, have the limitation that the position can only be changed to a restricted extent. Scenes comprising some billions of triangles require very high computational power and a very large memory, and can only be processed on fast and complex mainframes or by means of cluster solutions.

This is why software-based, dynamic, real-time ray-tracing systems cannot be realized with available staff-computer hardware. It is likely that for reasons of cost, the described cluster solution will remain restricted to special applications.

By contrast, the object of this invention is to propose a device with which ray tracing in dynamic, complex, three-dimensional scenes can be performed faster – preferably also in real time – in such manner that a photorealistic representation is obtained.

This object is established by a device according to claim 1, said device having at least one programmable ray-tracing processor in which are implemented:

- special traversing instructions and/or
- vector arithmetic instructions and/or
- instructions for establishing ray-tracing acceleration structures and/or
- at least one decision unit (mailbox), which prevents objects or triangles that have already been intersected by a ray cast during ray tracing from being intersected again by the ray.

The inventive device is organized in such manner as to allow a plurality of threads to be processed in parallel and a plurality of threads to automatically be processed synchronously. In addition, the device is provided with an n-level cache hierarchy and/or virtual memory management and/or a direct link to the main memory.

This device may preferably be realized in FPGA and/or in ASIC technology and/or another logic-based semiconductor technology or in discrete integrated logic, or in a combination of these technologies.

For a more detailed discussion of the decision unit, reference is made to Figure 2, from which it is evident that the list unit has been extended by a mailbox. When a ray is cast, this mailbox notes which objects or triangles the ray intersects and prevents any one triangle or object from being intersected more than once by the ray. As a result, fewer ray-object, i.e. ray-triangle, intersection computations need to be carried out, and this accelerates the computation. The mailbox may be seen as a kind of intersection-computation cache, which, unlike a memory cache, does not prevent memory requests to the memory but prevents intersection computations instead. Standard caching processes such as 4-way caches may be used to implement the mailbox.

Claim 2 relates to a device for the photorealistic representation of dynamic, complex, three-dimensional scenes by means of ray tracing, wherein said device has at least one special traversal unit, at least one list unit, at least one decision unit (mailbox) which prevents objects or triangles that have already been intersected by a ray cast during ray tracing from being intersected again by the ray, at least one intersection-computation unit, at least one unit for establishing acceleration structures, at least one transformation unit and/or at least one unit for solving linear equation systems, and wherein a plurality of rays or threads may be processed in parallel and a plurality of rays or threads may automatically be processed synchronously and an arbitrary number of dynamic-object levels may be realized in dynamic objects, and wherein the device is provided with an n-level cache hierarchy and/or virtual memory management and/or a direct link to the main memory.

The embodiment according to claim 3 differs from the embodiment according to claim 2 in that, in claim 3, a ray tracing processor already described in claim 1 has supplemented the at least one transformation unit and/or the at least one unit for solving linear equation systems and/or the at least one intersection-computation unit in claim 2.

This ray-tracing-based device for the photorealistic representation of three-dimensional moving scenes, in which device the acceleration structures and processes defined in the software have been implemented in the corresponding hardware structures, is primarily intended for real-time use.

For the realization of arbitrary, disordered dynamics in a scene, the acceleration structure has to be computed anew for each image of the image sequence. This means that large scenes involve a huge computational overhead, since the entire geometry of the scene has to be "handled". In such cases, the advantage of the logarithmic complexity is swallowed up by the size of the scene.

A solution – described in the paper "A Simple and Practical Method for Interactive Ray-Tracing" – to this problem is to subdivide the scene into objects and to allow the movement of these objects exclusively in their entirety. This necessitates two acceleration structures.

A top-level acceleration structure above the objects of the scene, and a bottom-level acceleration structure for each of the objects. The objects are positioned in the scene in the form of object instances.

The difference between an object and its instance is that an object instance consists of an object and a transformation. The transformation is an affine function that shifts the object to an arbitrary location in the scene. Affine transformations additionally permit scaling, rotating and shearing of objects. In the following, for the sake of simplicity, the term object will also be used for object instances unless there is a chance of confusion.

A ray is first traversed through the top-level acceleration structure (ray tracing through the scene) until a potential hit object (the object encountered by the ray) is found. The ray is now transformed into the object's local coordinate system and goes on to traverse in the bottom-level acceleration structure of the object until an intersection point with a primitive object is found. Primitive objects are objects possessing no sub-structures. In ray tracers, these are generally triangles and spheres.

This method works very well in practice, but only as long as the number of objects does not become too large, as the top-level acceleration structure has to be rebuilt in every image. Rebuilding this acceleration structure is necessary if the objects within are moved.

The invention presents a solution that recursively supports the above subdivision of the scene into objects. In other words, it is not restricted to objects that are primitive objects but also permits these objects to be made up of sub-objects which themselves may also consist of sub-objects, etc. Fig. 1 illustrates how a tree can be built up from several object levels. To start with, a leaf is modeled as a level 1 object. This leaf is now instantiated repeatedly and applied to a branch, thus creating another object, this time a level 2 object. These small branches can now be instantiated repeatedly again to form a larger branch, or tree, as level 3 object. It may be remarked that here, several object levels occur in objects, and that the representation of the scene is small on account of the same geometries being used a plurality of times.

The ray-casting process used according to claim 2 of the invention is as follows:

The ray is traversed through the top-level acceleration structure until a potential hit object is found. If the object is a primitive object, the ray-object intersection is computed. If the object is not a primitive object, the ray is transformed into the object's local coordinate system and there continues the traversal recursively.

An essential part of the process is the transformation of the ray into the object's local coordinate system, as a result of which, in principle, the positioning of the object is cancelled by the affine transformation. In other words, the transformed ray now sees the object as no longer transformed. This transformation step requires a very complex affine transformation of the ray's origin and direction; however, the complicated hardware unit needed for this purpose can also be used for other tasks. It transpires that the transformation unit can also be used to compute intersections with many kinds of primitive objects, to compute primary rays and to compute many kinds of secondary rays.

For the computation of primary rays, a similar camera transformation matrix is used as in the known rasterization processes. To start with, pre-primary rays of the type R=((0,0,0),(x,y,1)), that is, rays with the origin (0,0,0) and the direction (x,y,1), are defined, where x and y represent the coordinates of the pixel for which a primary ray is to be computed. For every camera position and alignment there is an affine transformation which transforms the ray R in such manner that it corresponds exactly to the incident direction of the camera pixel (x,y).

In order to compute the intersection with a primitive object, the ray is transformed into a space in which the primitive object is normalized. In the case of a triangle as primitive object, the ray is transformed, for example, into a space in such manner that the triangle has the form Δ_{norm}=((1,0,0),(0,0,0),(0,1,0)). This is illustrated in Fig. 10. This transformation can occur by means of an affine transformation. In contrast to the general case, the subsequent computation of the intersection with the normalized triangle is very easy to solve in hardware. If the transformation is selected such that the triangle normal is transformed onto the (0,0,1) vector in the triangle space, the scalar product of ray and triangle normal can be computed very easily in the triangle space, since the scalar product of the ray direction (xt,yt,zt) and the triangle normal (0,0,1) is precisely 0*xt+0*yt+1*zt = zt.

The transformation may also be selected such that only 9 floating-point numbers are needed to represent it; this is effected by imaging the triangle normal onto a suitable normal in the normalized triangle space. However, this rules out the opportunity of computing the scalar product in the normalized triangle space.

It is clearly evident that this normalized object transformation may also be used for other kinds of objects, for example spheres, planes, cuboids, cylinders and many other geometric shapes; it is merely necessary to create a different intersection-computation unit in each case.

A major advantage here is the fact that every kind of primitive object is represented identically in the memory, namely as an affine transformation which transforms into the normalized object space. This makes it easier to configure the memory interface of a hardware solution. The transformation into the normalized object space is referred to as normalized space transformation.

Shadow rays and specular reflections may be computed efficiently with the transformation unit by computing suitable transformations and suitable rays.

The transformation unit can also be used to transform normals (vectors which are perpendicular to a surface). This transformation of normals is important because some shading models require the geometry normal at the point of impact. However, this normal must be available in the world coordinate system, which is not necessarily the case in the above process. Much rather, the normal is initially only available in the local coordinate system of the object encountered. It has to be transformed from there back into the world coordinate system.

The transformation unit, however, also has a drawback. Since the affine transformations, which may be stored as matrices, have to be pre-computed both for triangles and for the objects of the scene, it is not so easy to change the position of the triangle vertices efficiently from frame to frame. In vertex shaders on modern graphic cards, however, this is possible. Vertex shaders are specialized programmable units that are optimized for computing movements from points in space.

To make this possible, it is necessary to do without the preliminary data computation. Accordingly, computing an intersection with a triangle then necessitates solving a linear equation system with three unknowns. True, this explicit method of solution requires more floating-point operations, but it is necessary in conjunction with vertex shaders. Accordingly, it may make sense to replace the above-described transformation unit with a unit that solves a linear equation system. This unit may be used, among other things, to intersect with triangles or to transform rays into the local coordinate system of an object.

The ray casting method used according to claim 1 of the invention is of a similar nature to the method described in claim 2, except that the transformation and the intersection computation are implemented by suitable instructions for the ray-tracing processor. Particularly attractive here is the possibility of using alternative methods to compute ray-object intersections. For example, testing for ray-triangle intersections with Plücker coordinates permits efficient use of vertex shaders.

A problem connected with very detailed scenes is undesired aliasing effects. These are encountered particularly when the object density is very high in one direction. It may then happen that a ray strikes a black triangle, for example, and, if the camera is moved minimally, a white triangle is suddenly struck. Effects of this kind cause temporal and local aliasing noise in the image. The reason for this is that ray tracing generally uses infinitely narrow rays and does not take into account the fact that the light influencing a pixel spreads out like a pyramid, and that the ray widens with increasing distance. In actual fact, all the objects within this ray pyramid should be included for the pixel-color computation, but this is not possible in a real-time system. A new, simplified form of cone tracing is helpful here. Instead of considering an arbitrarily narrow ray, the ray's acceptance angle is evaluated additionally. This makes it possible to compute the width of the ray in question, which will depend on the distance from the camera. If, during traversal, a cell is encountered that is largely overlapped by the ray, it may not be useful to continue traversal. At this location, it may be beneficial to use simplified geometry of the interior of the volume for computation purposes. The fact that perhaps a million triangles are contained within the volume can then be ignored. It is possible that these triangles merely form a mountain face, which, on account of the size of the ray, can also be approximated by a colored plane.

However, if the triangles form a perforated entity such as the Eiffel Tower, it is better to select the color of the constructive grid elements and a transparency value as approximation.

It is to advantage that simplified geometry representations like these can be supported in the acceleration structure. Figure 6 illustrates the idea using an octree as example. The simplified geometry of the boldly outlined volume, to which a node in the acceleration structure belongs, is shown. The ray overlaps almost the entire volume of the node, which means the simplified geometry can be used to compute the intersection.

An alternative method consists in storing the scene objects of the scene with different detail levels. This means that the objects are modeled using different resolutions or numbers of triangles, and that, additionally, detailed or simplified objects are used depending on the distance of the objects from the camera.

The fact that it is complicated to configure the above-described, permanently-wired, hardware-based ray-tracing pipeline programmably could be seen as a disadvantage thereof. Compared to a software-based ray-tracing approach, the hardware-based pipeline appears very rigid and specialized.

A CPU developed especially to suit the ray-tracing process is helpful here. This special ray-tracing processor consists of a standard CPU, for example a RISC processor, whose instruction set has been expanded by special instructions. Of particular importance is a traversing instruction which traverses the ray through an acceleration structure. Some stages of the process, furthermore, require complicated arithmetic operations that are carried out largely in three-dimensional space. It therefore makes sense to equip the CPU with a vector arithmetic unit, similar to today's familiar SSE2 instruction sets.

The CPU can be further optimized by exploiting the process's potential for parallel operations. For example, a plurality of threads (program runs) can be executed at once very effectively on a CPU, thus significantly enhancing CPU utilization and effectiveness. This applies particularly to waiting times for memory requests: if one thread makes a memory request, another thread can be executed during the request. Figure 5 shows a typical design for a CPU of this kind.

Since dynamic scenes require that acceleration structures be recomputed for every frame, the CPU's instruction set has to be expanded by special instructions for establishing acceleration structures. During the establishment of acceleration structures it is often necessary to decide whether or not an object is located in a given cell into which it should be sorted. A special unit that optimizes the establishment of acceleration structures can accelerate the necessary computation by making a very easily computed preliminary decision. As cells, boxes are often used whose 6 bounding surfaces are perpendicular to the x, y and z axes. A box of this kind can be characterized by its vertices, the definition of just two vertices in fact being sufficient. It is often possible to determine/decide whether the triangle is located in this box by simply comparing the coordinates of the vertices. If, for example, the triangle is located far to the left of the box in the x direction, the x coordinates of the triangle's 3 vertices will all be smaller than the smallest x coordinate of the box's vertices (see Fig. 13). Many other arrangements can be decided in the same way, for example, whether the triangle is located entirely within the box. If no decision is possible, complicated mathematical formulae such as the SAP (Separating Axis Theorem) have to be used. The decision as to whether a box overlaps with another box can likewise be made by means of vertex comparisons.

For purposes of further optimization, a decision unit is used that prevents objects or triangles that have already been intersected by a ray cast during ray tracing from being intersected by the ray again. This is effected by expanding the list unit by a mailbox, as shown in Fig. 2. When a ray is cast, this mailbox notes which objects or triangles the ray intersects and prevents any one triangle or object from being intersected more than once by the ray. As a result, fewer ray-object, i.e. ray-triangle, intersection computations need to be carried out, and this accelerates the computation. The mailbox may be seen as a kind of intersection-computation cache, which, unlike a memory cache, does not prevent memory requests to the memory but prevents intersection computations instead. Standard caching processes such as 4-way caches may be used to implement the mailbox.

The units making up the ray tracing architecture require a very high memory bandwidth, in other words, very high data volumes have to be transmitted per unit of time. Normally, this can only be realized by connecting up very large numbers of memory chips in parallel. However, the necessary memory bandwidth can also be obtained by means of a suitable arrangement of multiple cache levels (n-level caches). Of the utmost importance here is a property of ray tracing known as coherence. Coherence denotes the fact that rays passing through similar areas of the 3D space access almost the same data in the acceleration structure and, accordingly, also the same objects. If this property is exploited, high cache hit rates can be obtained. That means it is highly probable that the required data will be found again in the cache, thus obviating the time-consuming necessity of downloading them from the main memory. As shown in Figure 4, the caches themselves are arranged, for example, in a binary tree so as to serve a plurality of ray-tracing units.

Used in conjunction with a 3D display, the inventive device can naturally also be used for the photorealistic, three-dimensional, real-time representation of complex moving scenes. Depending on the display technology used, there are three image-output variants.

First, a configuration in which two images containing the stereo imprint are represented alternately, horizontally staggered, on a display in a time-division multiplex mode.

Second, a configuration in which two horizontally staggered images representing the stereo imprint are represented on a display in alternating vertical strips containing the picture information for the two images.

Third, a configuration in which the two horizontally staggered images are represented simultaneously on two separate displays or in a time-division multiplex mode.

The two horizontally staggered images, one to be assigned to the right eye and one to the left eye, are each rendered visible to one eye only by means of suitably spatial display arrangements or through use of image separation means (e.g. shutter spectacles, strip-like Fresnel prisms/lenses and polarization filters). The 3D displays to be used, and their video signal control requirements, are in keeping with the prior art and are not described in detail. Further information on the prior art relating to 3D displays may be taken from the following publications, which are cited as examples : Computer Graphics / Addison-Wesley ISBN 0201848406, DE 4331715,DE 4417664, DE 19753040, DE 19827590, DE 19737449

The uses of computer-animated, photorealistic, real-time, three-dimensional moving scenes and images range from the representation of three-dimensional CAD data and of medical and technical-analytical data, through film animation and use in flight simulators and driving simulators, to so-called "home" applications in computer games with complex real-time graphics.

The same processes may additionally be used – without further modifications to the functional configuration - for non-photorealistic image generation (e.g. line drawings or the representation of comic stills). It is likewise possible, again without the need for any technical modifications, to perform computations that are not generally associated directly with image computation. Examples here include collision detection for geometric objects and the discrete solving of numerical problems. None of the applications described are restricted to the interactive sector and all can also be used offline – for example for cinema-film computations or very complex physical simulations – without any modifications to the process or the device.

The functional realization and the hardware implementation of ray tracing is effected in complex and fast logic technologies, the implementation of which being effected both as permanently-wired digital logic in the form of discrete digital logic comprising individual standard ICs, or customer- and application-specific ICs such as ASICs, or in the form of complex, programmable logic devices / logic circuits, for example CPLD and FPGA technologies with or without a CPU core, or a combination of these technologies.

The exemplary embodiment, described below, of the invention describes the ray-tracing unit of a computer graphic card in which the hardware implementation of the ray-tracing process may be effected, by way of example, in a freely programmable FPGA logic device, in ASIC technology, or in a permanently wired custom chip.

In so far as processes and operational sequences are described, these are to be realized purely in hardware. This means that appropriate logic units and hardware-implemented arithmetic units must be configured.

The standard control electronics for controlling the data display (cathode ray tube, TFT, LCD or plasma monitor), and the timing thereof, are in keeping with the prior art, are assumed to be known and are not subject matter of the description. An interface between the image memory of this standard equipment and the implementation, as according to the invention, of the ray-tracing process is described.

The description comprises two parts: First, the ray-casting pipeline (in short, RCP) is described. This is the central entity of the design; it traverses the rays through the scene and sends back intersection data.

In the second part, an optimized ray-tracing architecture for the ray-casting pipeline is described; a plurality of these ray-casting pipelines operate together within this architecture.

Figure 2 illustrates the ray-casting pipeline (RCP), which consists of several subunits. The traversal unit traverses the ray through a suitable acceleration structure, preferably a kD tree. The ray is traversed until it reaches an area of the scene in which potential hit objects are located. The objects in this area are stored in a list which is processed by the list unit. The list unit contains a mailbox, which, when a ray is cast, notes which objects or triangles the ray intersects and prevents any one triangle or object from being intersected more than once by the ray. As a result, fewer ray-object, i.e. ray-triangle, intersection computations need to be carried out, and this accelerates the computation.

These objects must now be tested for a possible intersection. If there is no valid intersection, traversing must be continued. The list unit sends the potential hit objects that have not yet been processed to the matrix-loading unit one after the other. The matrix-loading unit loads the affine transformation belonging to the object. This affine transformation can be represented by a 4x3 matrix. The matrices may be object-space transformation matrices or matrices that transform into the normalized space of a primitive object. After the matrix-loading unit has stored the matrix in the transformation unit, the ray-casting unit sends the rays through the transformation unit.

Following the transformation, two scenarios are now possible: First, the object involved may be an object that contains sub-objects. If this is the case, the rays return to the traversal unit and the transformed ray continues to be traversed in the object. If, however, the object is a primitive object, the ray goes on directly into the intersection-computation unit, which intersects the ray with the normalized object. As described earlier, the intersection-computation unit is able to support a plurality of normalized objects (triangles, spheres, etc.)

The computed intersection data are collected in the intersection unit. The intersection unit provides a reverse channel to the traversal unit, so that this can recognize whether already-valid intersection data are available.

The traversal unit, list unit and matrix-loading unit are the only components of the ray-casting pipeline which access external memories. The traversal unit accesses the acceleration structure, the list unit accesses lists of object addresses and the matrix-loading unit accesses affine transformations in the form of matrices. In order to ensure the necessary memory bandwidth, all three units are connected to the main memory via a dedicated cache.

A simplified version of the ray-casting pipeline, showing just the most important components, is illustrated in Figure 12: the traversal unit, which traverses the rays through the acceleration structure, the list unit, which processes the lists, the transformation unit, which applies the loaded transformation to the rays, and the intersection-computation unit, which intersects the transformed ray with the normalized object.

The ray-casting pipeline is embedded in a suitable ray-tracing architecture, as shown in Figure 3. The drawing shows 4 ray-casting-pipeline units, each with 3 caches. Two of these units, in each case, are connected with a shading unit. These shading units use the ray-casting-pipeline units to compute the colors of the image pixels. To this end, the shading unit shoots primary rays, processes the hit information sent back by the ray-casting pipeline, and shoots secondary rays, for example to light sources. The shading units may be in the form of permanently wired hardware or the programmable ray-tracing processors described later.

The shading units possess a channel to the transformation unit of the ray-casting pipeline. This is used to load camera matrices and matrices for secondary rays and thus to minimize the computational overhead for the shading unit. It is beneficial if the shading units each have a separate texture cache and shading cache. The shading cache contains shading information on the scene geometry, for example colors and material data. The texture cache is connected with the texture memory and enables the shading units to access textures. It is to advantage if each shading unit has its own local frame buffer in which it stores the colors and brightness values of the pixels currently being processed / computed. It is additionally beneficial if a z buffer is provided, as this is needed for the subsequently described connection to the standard rasterization hardware.

Once the color of a pixel has been fully computed by the shading unit, this color can be written into the global frame buffer via the optional tone-mapping unit. The tone-mapping unit applies a simple function to the color in order to image it in the 24-bit RGB space. The geometric depth values (z values), too, which are stored in the optional, local z buffer, can now be transferred into the optional global z buffer.

However, the color and the new z value are only written into the frame buffer or z buffer if the z value that was in the z buffer before is greater. This measure ensures that pixels are only written if they lie geometrically ahead of values already computed by the rasterization hardware or other ray-tracing passes. Through this arrangement, it is possible to combine the ray-tracing hardware with standard rasterization hardware operating on the same frame buffer / z buffer, which also constitutes the interface to this standard hardware.

To further increase the system power, additional optional shading units may be connected up in parallel with the associated ray-casting pipelines and caches. The power-boosting effect lies in the broadening of the data structure and the processing structure.

It is unlikely that the entire scene will fit into the local main memory of a ray-tracing chip. To solve this problem, the local main memory may be used as a large cache that caches sizable blocks of the scene. The actual scene is located elsewhere and is downloaded via DMA when needed. This virtual memory management process makes it possible to visualize large scenes that do not fit into the ray-tracing chip's main memory.

Photon mapping is a standard technique in which virtual photons are shot from the light sources into the scene and are collected on the surfaces of the scene. The light distribution of the scene can thus be simulated. This applies above all to caustics. If photons are shot, an acceleration structure – a kD tree, for example – is built up over the photons. Now, an image of this computed photon light distribution can be effected by visualizing the scene with standard ray-tracing techniques and incorporating the incident light intensity at every point of impact into the color computation in such manner that the energies of all the photons striking in the vicinity of this point are added up. This entails searching for all the neighboring photons in the photon acceleration structure. The traversal unit can help with this task by traversing a volume instead of traversing along a ray. All the neighboring photons can be processed in this way, for example by adding up their energies.

As a rule, ray-tracing processes operate with infinitely narrow rays, and this leads to sampling artifacts (aliasing). A much better approximation is obtained by using ray cones instead of rays. The light that illuminates a camera pixel does not come from a discrete direction but from a kind of pyramid, which can be approximated very well by a cone. Traversal of this space volume, described by a ray cone or ray pyramid, from front to back can likewise be performed, as a special function, by the traversal unit. Figure 6 shows a two-dimensional drawing of a ray cone.

The signal conditioning for control of the display and the generation of the timing for the display or monitor may be effected in known manner by means of the optional rasterization hardware or implemented by suitable functional units if beneficial to the desired application. For example, the basic functions of standard rasterization hardware may be combined with the hardware-implemented ray tracing processes and functions to form a very powerful real-time hardware architecture.

A second exemplary embodiment of the invention is based on the configuration and use of freely programmable ray-tracing CPUs or ray-tracing processors, which are program-controlled to carry out the special ray-tracing functions and processes described in the invention. Thanks to appropriate logic parallelism and function parallelism, only a few – preferably one or two – CPU tact cycles are needed here to process the individual functions.

In so far as internal algorithms, processes and operational sequences of the ray-tracing processors are described, these are to be established using a hardware description language such as HDL, VHDL or JHDL and transferred to the hardware in question. The hardware implementation of the ray-tracing processors, with the implemented processes and functions, may be effected, by way of example, in a freely programmable FPGA logic device, in ASIC technology, in a combination of digital signal processors and FPGA/ASIC, or in a permanently wired custom chip.

This means that appropriate logic units and hardware-implemented arithmetic units must be configured, whose individual functions can be retrieved by way of program control. Depending on the complexity of the hardware technology employed, one or more ray-tracing processors can be realized per chip, with or without additional logic functions.

Ray-tracing processors are fully programmable computing units which are engineered to carry out vector arithmetic instructions and special ray-tracing instructions, such as “traversing” and "establishment of acceleration structures". The configuration may incorporate additional functional units or else make use of already available functional units plus a few additional logic devices if required. For example, traversing may be effected by means of special functional units or by expanding the available arithmetic functional units by a few logical functional units.

As shown in Figure 11, several of the ray-tracing processors illustrated in Fig. 5 are connected in parallel. The memory interface is formed by a cache hierarchy that provides the necessary memory bandwidth. This procedure is efficient due to the strong coherence of adjacent rays. With the ray-tracing-processor system, each ray-tracing processor processes precisely one image pixel or one packet comprising several pixels. Here, the computation of each individual pixel corresponds to one computation thread. A plurality of these threads are bundled to form a packet and are processed synchronously as a whole. Synchronously processed threads are characterized in that they always jointly perform the same instruction. This makes it possible to create efficient hardware and to carry out memory requests per entire packet, not just individually for each thread. Particularly this reduction in the number of memory requests is a substantial advantage of processing packets of threads.

The thread generator of Figure 11 creates threads which are processed on the ray-tracing processors. The thread generator can also be programmed by the ray-tracing processors. Special functions for scanning the image pixels in a cache-coherent manner (for example, Hilbert curve) make it possible to supply the ray-tracing processors optimally with coherent threads. This reduces the required memory bandwidth. Via a DMA interface, the thread generator also has access to the main memory. It is accordingly also possible to create the start values for the individual threads in a preliminary processing step and to write them into the memory, so that new threads can be generated from this information later on.

Pixels are processed by means of a software program that runs on the ray-tracing processor. This software program describes a recursive process sequence for computing the color value of a pixel, even if the hardware works on packets of rays. The packet management is accordingly transparent to the programming model.

Figure 5 shows a typical structure for a ray-tracing processor. Visible is a standard processor core (RISC core) to which two special co-processors are connected in parallel. Each co-processor has its own register. However, by means of special instructions, it is also possible to transfer the contents of these registers from one co-processor into another. The traversal core is a special co-processor that is able to efficiently traverse the rays through an acceleration structure. To do this, it needs a special memory interface to the nodes of the acceleration structure (node cache). The vector arithmetic core is a special co-processor that is able to efficiently perform operations in 3D space. The vector additions, scalar multiplications, cross products and vector products needed by every ray-tracing software can be computed rapidly with this unit. The vector arithmetic unit requires access to a special cache which enables it to load whole vectors in a single tact.

The semantics of the traversal instruction could look as follows: the ray-tracing CPU writes into special registers the origin and direction of the ray and the address of the acceleration structure to be traversed. A specific instruction is now called, which starts a specific traversal unit. This unit traverses the acceleration structure and sends all nodes containing objects to a list unit; the latter may contain a mailboxing mechanism that has already been described in the first exemplary embodiment of the invention. For every object, the CPU now executes a small program that intersects the ray with this object. If a valid ray-object intersection is found, the program that shot the ray takes over again.

The individual instructions or logic functions that are hardware-implemented in the ray-tracing processor contain the same algorithms as have already been described for the permanently wired, first embodiment of the invention. In addition to these, however, this instruction set may be extended by supplementary, permanently wired instructions and functions, the activation of which is again program-controlled. Thanks to special traversing operations, vector arithmetic and parallel processing of a plurality of threads on a ray-tracing processor, the necessary computational power for the real-time application is ensured while, at the same time, the effects of memory latencies (memory-request waiting times) on the system speed are minimized or even become irrelevant.

Once the pixel color has been fully computed, it can be written into the frame buffer. Additionally, the distance may be written into the z buffer. A connection with the rasterization hardware, and corresponding representation on a display, is thereby possible.

Claim 10 describes the use of a supplementary space-dividing data structure, which does not store or reference any geometric objects but contains spatial influences or material-modifying parameters. Spatial influences of this kind may include fog, haze, dust particles or hot smoke; hot smoke, for example, may also cause a modification to the scene visible through the smoke volume. Further spatial influences are light sources or material-modifying auras of the kind that may be used, for example, to represent imaginary entities. The use here of the space-dividing data structure permits a considerable reduction in the computational overhead, as only those influences need to be taken into account that are spatially located such that they are able to have an influence.

Claim 11 describes a further development that makes it possible to process three-dimensional scenes which are not, or not exclusively, made up of triangles, and that, where necessary, transforms other geometric primitives into triangles or into an easily-processed intermediate format. To obtain the additional functionality here, the hardware is either provided with additional functional units, or already available units implement the functionality.

Claim 12 describes a further development that makes it possible to compute, for each ray, not only the closest but, additionally, other ray-object intersections as the result of a ray-processing operation. It is beneficial here if the results are sorted according to the distance from the ray's origin. The maximum number of intersections per result may be defined as a constant or be described for each ray as a function of parameters of the intersected objects. This technique may be used to compute rays passing through transparent objects more efficiently.

Claim 13 describes a further development that is able to count, with additional and/or the available functional units, how often a specific element was used in computing an image. The elements to be counted here may differ greatly, and might possibly be classified via the address in the memory or via an element's ID. These elements include:

dynamic and geometric objects, partial or complete material descriptions, elements or subgroups of a space-description data structure, programs or program functions, individual memory cells and entire memory pages or memory areas.

Claim 14 describes an extension of the functionality for computing the space-description data structures for partial or complete three-dimensional scenes, where additional parameters for each dynamic object or dynamic sub-object or geometric object influence the manner in which the space-description data structure is computed. Influences of this kind may be, for example, that an object specifies that it possesses no geometric objects in given space volumes. This permits more efficient computation of the space-description data structure and, additionally, a further reduction in the per-image computational overhead.

Description of the drawings

Figure 1 illustrates how a tree can be built up from several object levels. To start with, a leaf is modeled as a level 1 object 101. This leaf 101 is now instantiated repeatedly and applied to a branch 102, thus creating another object, this time a level 2 object. These small branches 102 can now be instantiated repeatedly again to form a tree 103 as a level 3 object.

Figure 2 illustrates the ray-casting pipeline 200, along with the most important data paths, and the interface to the caches. The arrow 201 goes to the shading unit, while the arrow 202 comes from the shading unit. Below is a key to the other reference numerals:

203:	Traversal unit
204:	List unit, which contains a mailbox
205:	Matrix-loading unit
206:	Ray-casting unit
207:	Transformation unit
208:	Intersection-computation unit
209:	Intersection unit
210:	Node cache
211:	List cache

212:	Matrix cache

Figure 3 illustrates the top-level diagram of an exemplary implementation of the invention. The 4 ray-casting pipelines (RCP) are connected with the separate memories for nodes, lists and matrices via a cache hierarchy. In this example, two of these RCPs, in each case, are connected with a shading unit that has access to a local frame buffer and z buffer. Via these, color values are written by means of a tone-mapping unit into a global frame buffer, and depth values are sent direct to a global z buffer. Prior-art rasterization hardware (rasterization pipeline) can be connected to the z buffer and the frame buffer.

Below is a key to the reference numerals:

301:	Rasterization hardware
302:	Frame buffer
303:	Video out
304:	Tone-mapping unit
305:	Z buffer
306:	Frame buffer 1
307:	Z buffer 1
308:	Texture cache 1
309:	Shading cache 1
310:	Shading unit 1
311:	Node cache 1
312:	List cache 1
313:	Matrix cache 1
314:	RCP 1
315	Node cache 2
316:	List cache 2
317:	Matrix cache 2
318:	RCP 2
319:	Frame buffer 2
320:	Z buffer 2
321:	Texture cache 2
322:	Shading cache 2
323:	Shading unit 2
324:	Node cache 3
325:	List cache 3
326:	Matrix cache 3
327:	RCP 3
328	Node cache 4
329:	List cache 4
330:	Matrix cache 4
331:	RCP 4

Figure 4 illustrates the cache infrastructure that provides the necessary internal memory bandwidth in the chip. The infrastructure in the drawing is a binary n-level cache, but other hierarchical structures are also conceivable.

Below is a key to the reference numerals:

401:	Node cache 1
402:	Node cache 2
403:	Node cache 3
404:	Node cache 4
405:	Node cache 5
406:	Node cache 6
407:	Node cache 7
408:	Node memory
409:	List cache 1
410:	List cache 2
411:	List cache 3
412:	List cache 4
413:	List cache 5
414:	List cache 6
415:	List cache 7
416:	List memory
417:	Matrix cache 1
418:	Matrix cache 2
419:	Matrix cache 3
420:	Matrix cache 4
421:	Matrix cache 5
422:	Matrix cache 6
423:	Matrix cache 7
424:	Matrix memory
425:	Texture cache 1
426:	Texture cache 2
427:	Texture memory
428:	Shading cache 1
429:	Shading cache 2
430:	Shading cache 3
431:	Shading memory

Figure 5 illustrates the exemplary embodiment of a ray-tracing CPU. Below is a key to the blocks:

501:	Load instruction
502:	Instruction memory
503:	RISC core
504:	Cache
505:	Traversal core
506:	Node cache
507:	Vector arithmetic core
508:	Vector cache

Figure 6 illustrates an example of simplified geometry in the octree nodes. The reference numeral 601 denotes the ray cone, and the reference numeral 602 an instance of “simplified geometry”.

Figure 7 illustrates an example of the uniform-grid acceleration structure at a simple scene. For the sake of simplicity, the space is only shown in 2D. The reference numeral 701 denotes the ray.

Figure 8 illustrates an example of the kD-tree acceleration structure at a simple scene. For the sake of simplicity, the space is only shown in 2D. The reference numeral 801 denotes the ray.

Figure 9 illustrates an example of the octree acceleration structure at a simple scene. For the sake of simplicity, the space is only shown in 2D. The reference numeral 901 denotes the ray.

Figure 10 illustrates the global to normalized object space transformation for a triangle. To be seen is a ray 1001 and a triangle 1002. A transformed ray 1004 and a normalized triangle 1005 are generated by an affine transformation represented by the arrow 1003. The global space is represented by the left coordinate system and the normalized triangle space by the right coordinate system.

Figure 11 illustrates an exemplary embodiment of the invention, which is based on special ray-tracing processors. Below is a key to the blocks:

1101:	Rasterization hardware
1102:	Thread generator
1103:	Z buffer / frame buffer
1104:	Video out
1105:	Ray-tracing processor 1
1106:	Ray-tracing processor 2
1107:	Ray-tracing processor 3
1108:	Ray-tracing processor 4
1109:	Cache 1
1110:	Cache 2
1111:	Cache 3
1112:	Cache 4
1113:	Cache 5
1114:	Cache 6
1115:	Cache 7
1116:	Main memory

Figure 12 illustrates a simplified version of the ray-casting pipeline shown in Figure 2. Below is a key to the blocks:

1201:	Traversal unit
1202:	Node cache
1203:	List unit
1204:	List cache
1205:	Transformation unit
1206:	Matrix cache

1207: Intersection-computation unit

The arrow 1208 comes from the shading unit.

Figure 13 illustrates a simple case in which evaluation and comparison of the vertices of a triangle and the vertices of a box permit a decision as whether the triangle and the box overlap. If the x coordinates of the triangle are smaller than the smallest x coordinate of the box, the triangle is located outside the box. The triangle is denoted by the reference numeral 1301, the box by the reference numeral 1302.